Worksheet 2 (part 1)

Marking scheme

Part 1

- Task 1 - 30%
- Task 2 - 30%
- Task 3 - 40%

All code should be submitted via a gitlab.uwe.ac.uk repo.

Each task should be documented in a README.md in the main repo, including screen shots of the code running, and details of how your implementation works. Additionally, the code should be structured using header files, along with comments throughout the code.

After completing this worksheet you should be familiar with:

- Defining different types of explicit dynamic memory allocation
- C++'s use of pointers
- C++ memory allocators

The deadline for completing this worksheet is 5th December, 2023, by 2pm.

> IMPORANT: All code must be submitted via gitlab, failure to provide a link to Gitlap repo will result in a mark of 0. No email or other form of submission is acceptable.

Memory allocation

In this worksheet, we will look at how to create our own heap allocator from scratch instead of relying on an existing allocator. We will discuss different allocator designs, including a simplistic bump allocator and a basic fixed-size block allocator, and use this knowledge to implement an allocator with improved performance.

The responsibility of an allocator is to manage the available heap memory. Within an operating system this could be to manage memory for a process, allocating pages via the `valloc` system call, for example. An applications runtime, such as the standard C or C++ library, build on these capabilites to provide heap allocation at the application layer. An allocator can often return unused memory on alloc calls and keep track of memory freed by dealloc so that it can be reused again. Most importantly, it must never hand out memory that is already in use somewhere else because this would cause undefined behavior.

Apart from correctness, there are many secondary design goals. For example:

- the allocator should effectively utilize the available memory and keep fragmentation low
- it should work well for concurrent applications and scale to any number of processors
- For maximal performance, it could even optimize the memory layout with respect to the CPU caches to improve cache locality and avoid false sharing.

These requirements can make good allocators very complex. For example, jemalloc has over 30K lines of code. This complexity is often more than necessary in allocators used in operating system kernels or in game engines, where general purpose memmory management is not necessary. Instead, correctness and performance are key requirements.

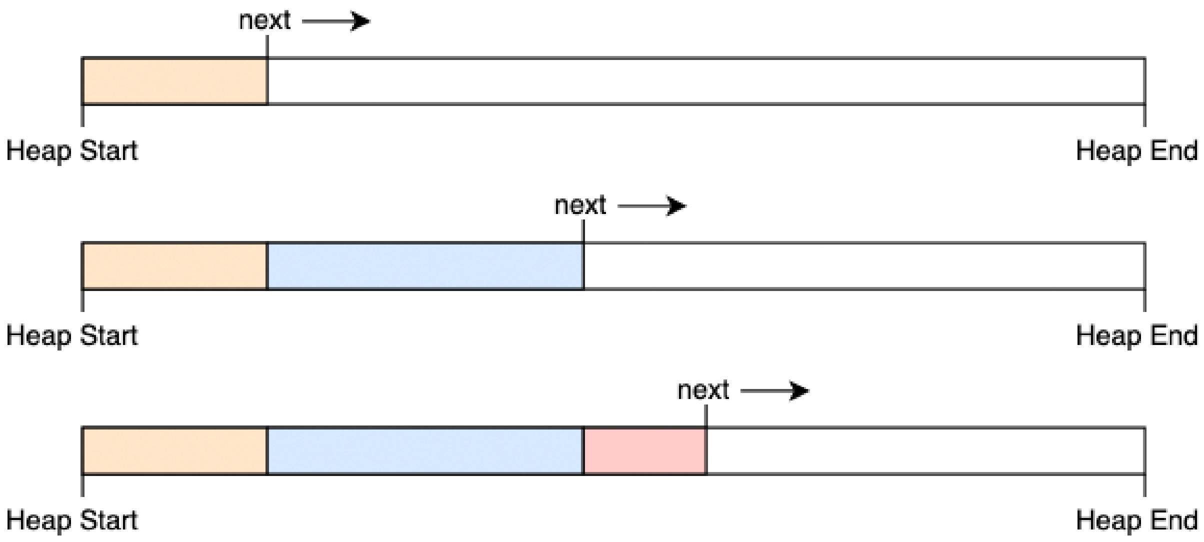
In this worksheet, we look at three possible allocator designs, considering their advantages and drawbacks.

Task 1 - Bump Allocator

The most simple allocator design is a bump allocator (also known as stack allocator). It allocates memory linearly and only keeps track of the number of allocated bytes and the number of allocations. It is only useful in very specific use cases because it has a severe limitation: it can only free all memory at once.

A Bump allocation is a super fast method for allocating objects. The allocator mantains a chunk of memory and a "bump pointer" within that memory. Whenever an allocation request is made for an an object, we do a quick test that to verify that there is enough capacity left in the chunk, and then, assuming there is enough room, the allocator moves the bump pointer over by `sizeof(object)` bytes and return the pointer to the space just reserved for the object within the chunk.

For example, at the beginning, a next`` is equal to the start address of the heap. For each allocation,next“‘ is increased by the allocation size so that it always points to the boundary between used and unused memory:



The `next` pointer only moves in a single direction and thus never hands out the same memory region twice. When it reaches the end of the heap, no more memory can be allocated, resulting in an out-of-memory error on the next allocation.

A bump allocator is often implemented with an allocation counter, which is increased by `1` on each `alloc` call and decreased by `1` on each `dealloc` call. When the allocation counter reaches zero, it means that all allocations on the heap have been deallocated. In this case, the next pointer can be reset to the start address of the heap, so that the complete heap memory is available for allocations again.

Design a bump allocator class, which should create space for the heap on creation, and provide at least the methods:

- `alloc`, which given a type allocates space for that N number of objects, returning a pointer for the allocation, or, if there is not enough memory, returns a `nullptr`.
- `dealloc`, free an allocation. (As the bump allocator only allocates upwards, the heap can only free memory when all memory allocations are freed.)

Your allocator should be type save, i.e. the allocation fun should return a pointer to type of data being allocated.

> Hint: a template member function might help in that case.

Write some examples to demostrate your allocator. These examples should demostrate at least the following capabilities of your allocator:

- allocate objects of different sizes and the use of the allocated memory
- allocate fails, returning `nullptr`, when not enough memory

- allocator resets when deallocations matches the number of (successful) allocations

Extend your allocator class to enable the user to specify how much space is allocated to the bump allocator on creation. This should be statically defined.

Document in your repo's readme.

Task 2 - Unit Tests

For this task you need to develop unit tests to validate your allocator. For this we will use the framework simpletest. You should add simpletest as a submodule to your git repo for this worksheet.

```
git add submodule repo-url
```

Read through the documentation for in paraticular have a look at the repo https://github.com/kudaba/simpletest_test. Write unit tests for your bump allocator. As an example, here is a short program that peforms a unit test for my bump allactor:

```
include "bump.hpp"

#include <simpletest.h>

using namespace std;

char const * groups[] = {
    "Bump",
};

DEFINE_TEST_G(BumpTest1, Bump)
{
    bump<20 * sizeof(int)> bumper;

    int * x = bumper.alloc<int>(10);
    TEST_MESSAGE(x != nullptr, "Failed to allocate!!!!");

    int * y = bumper.alloc<int>(10);
    TEST_MESSAGE(y != nullptr, "Failed to allocate!!!!");

    int * z = bumper.alloc<int>(10);
    TEST_MESSAGE(z == nullptr, "Should have failed to allocate!!!!");
}
```

```
int main() {
  bool pass = true;

    for (auto group : groups) {
        pass &= TestFixture::ExecuteTestGroup(group, TestFixture::Verbose);
  }

    return pass ? 0 : 1;
}
```

Document your bump allocator's unit tests in the README.md.

Task 3

Begin by reading the following blog post:

- Always Bump Downwards

Spend sometime thinking about what this blog is proposing and how your implementation works, does it bump up or down?

This task, as you probably guessed, is about optimizing your allocator to always bump down, but before doing any form of optimization it makes sense to first have some benchmarks. A benchmark is one or more programs that test the performance of your system. In this case we want to write a benchmark to test the performace of your allocator for different types of allcoation, e.g. lots of small allocations, lot of big allcoations, lots in between, and a selection of different sizes.

Your benchmark will need to time pieces of code. For this you should use C `<ctime>`, which supports different types of clocks, or (recommended) C++'s `<chrono>`.

Here is a basic example:

```
#include <chrono>

/* Only needed for the sake of this example. */
#include <iostream>
#include <thread>

// prototype of function we want to benchmark
void some_function();

int main()
```

```
{
    using std::chrono::high_resolution_clock;
    using std::chrono::duration_cast;
    using std::chrono::duration;
    using std::chrono::milliseconds;

    auto t1 = high_resolution_clock::now();
    some_function();
    auto t2 = high_resolution_clock::now();

    /* Getting number of milliseconds as an integer. */
    auto ms_int = duration_cast<milliseconds>(t2 - t1);

    /* Getting number of milliseconds as a double. */
    duration<double, std::milli> ms_double = t2 - t1;

    std::cout << ms_int.count() << "ms\n";
    std::cout << ms_double.count() << "ms\n";

    return 0;
}
```

Again take sometime to read over the Chrono documentation and some examples.

Design and implement a small benchmark library that report the time of a given function. Your library should be able to take a function of the type:

```
void function(void)
```

Extra marks if your benchmark can handle functions with different argument numbers and types.

For this you can use variadric templates (You could also use C macros, but I do not recommend this.):

- blog
- video

Design and implement a small benchmark suite, using your library, for your allocator.

Implement two versions of your bump allocator, one that bumps up and one that bumps down and finally, benchmark each version of your allocator.

Don’t forget to document both your benchmark implementation and its findings.